A PLATFORM INDEPENDENT BINARY INSTRUMENTATION METHOD

BACKGROUND

1. FIELD

The present invention relates generally to the binary code instrumentation domain, and, more specifically, to methods of designing platform-independent binary instrumentation systems.

2. DESCRIPTION

The process of binary instrumentation is widely used in software analysis, correctness checking, debugging, and performance monitoring, as well as in other areas where it has proved to comprise an efficient means of gaining control over a program being analyzed.

The idea behind binary instrumentation is to modify a compiled executable module prior to its execution to enable further analysis and/or debugging. Operating on binary executables eliminates the computational complexity normally associated with the processing of source codes, facilitates the creation of universal instrumentation and analysis systems, and enables dynamic instrumentation to be performed at runtime.

Many of the state-of-the-art binary instrumentation systems are highly dependent on processor instruction coding and other low-level system properties, and thus have to employ assembly language and very often are designed to operate on a limited set of software platforms or hardware architectures.

In addition, using an assembly language to implement platform-dependent code decreases the ease of debugging and support for the entire instrumentation system and sometimes prevents portability.

Therefore, a need exists for the capability to provide higher portability and manageability for binary instrumentation systems.

BRIEF DESCRIPTION OF THE DRAWINGS

The features and advantages of the present invention will become apparent from the following detailed description of the present invention in which:

Figure 1 is a diagram illustrating an exemplary runtime binary instrumentation system and relations between an original function, an instrumenting block, and an interceptor function;

Figure 2 is a diagram illustrating the layout of data structures employed in the process of binary instrumentation in accordance with an embodiment of the present invention;

Figure 3 is a flow diagram illustrating the instrumenting function operation according to an embodiment of the present invention;

Figure 4 is a flow diagram illustrating the interceptor function operation in accordance with an embodiment of the present invention.

DETAILED DESCRIPTION

An embodiment of the present invention is a method of implementing platform-independent binary instrumentation systems. The independence of specific hardware platforms may be achieved by employing multiple copies of the same procedure implemented in a high-level programming language as a substitution (interceptor) function that receives control prior to an original (intercepted) function. Additional means are provided to dynamically adapt to system memory conditions and the number of original functions to intercept.

Reference in the specification to "one embodiment" or "an embodiment" of the present invention means that a particular feature, structure or characteristic described in connection with the embodiment is included in at least one embodiment of the present invention. Thus, the appearances of the phrase "in one embodiment" appearing in various places throughout the specification are not necessarily all referring to the same embodiment.

The following definitions may be useful for understanding embodiments of the present invention described herein.

Binary Instrumentation is a process of modifying binary executable code in order to intercept particular regions of execution (e.g., specific functions) and transfer control to an external program (or a program code embedded in the binary executable being instrumented) to perform necessary operations.

An original function is an automatically determined or user-specified functional block (code region) identified by its starting address within a program to be instrumented.

An interceptor function is a functional block that gains control each time an original function is invoked, identifies the original function to call, optionally performs registration or analysis operations, and finally passes execution to the original function.

An instrumenting function is a functional block used to establish correspondence between the starting addresses of original and interceptor functions (in order to enable transfer of control to a corresponding interceptor function prior to executing an original function).

Figure 1 provides an example of a runtime binary instrumentation system.

According to Figure 1, in one example client module 12 may register a function address with service module 10 for further service during the normal flow of operation. Instrumenting module 14 may intercept the registration request and substitute the original function address with an address of a selected interceptor function copy 16 chosen in accordance with an

embodiment of the present invention. A runtime instrumentation system was chosen as the most complex case when the total number of functions to instrument is not known in advance. A more popular case of instrumenting a stand-alone executable module is similar to one described in Figure 1 with the only exception being that all initial addresses of functions to be intercepted pertain to one target module and may be pre-computed, and the information of the total number of such addresses may be used to optimize (or simplify) memory allocation. Embodying the platform-independent binary instrumentation method described herein may help seamlessly handle both cases.

Figure 2 depicts the general layout of data structures that may be used by the instrumenting module to establish an efficient mapping between the original and interceptor functions according to embodiments of the present invention. An embodiment of the present invention may utilize the same interceptor function code for all functions being intercepted but (in order to differentiate between multiple function instances) the interceptor function code may be copied to a different location (so that the execution is always transferred to different addresses for different intercepted functions). All such copies of the interceptor function may be organized in contiguous memory regions 22, normally allocated at run-time. This allows the instrumentation system to dynamically adapt to the number of functions to be instrumented (as it may not be known in advance).

In order to control the distribution of the above mentioned regions, a region descriptor table (RDT) is provided. The region descriptor table contains a plurality of region descriptors 20 which, in their turn, comprise at least the addresses of the beginning and the end of a region (the latter address may be substituted by the length of the region depending on what is preferable for a particular system architecture) and a reference (or an offset called a function map offset herein) to an intercepted function address table 24. The intercepted function address table contains addresses of all intercepted functions.

Figure 3 is a flow diagram illustrating the instrumenting function operation according to an embodiment of the present invention. Upon the initial request to instrument (intercept) a function or a code block, it may be determined if a current region is full or if no region is allocated at block 100. If the current region is allocated or no region is allocated, then a binary instrumentation system (as shown in Appendix A, for example) may allocate (at block 102) a memory region of a pre-defined (or pre-calculated) length and place a copy of the interceptor function body (at block 104) into that region. The corresponding region descriptor may be updated with appropriate addresses and a current offset to the intercepted function address table (IFT) at block 106. Then, the current element of the IFT may be initialized with the address of the function being intercepted (original function) at block 108. In case it is determined to be

more efficient to have the memory region pre-initialized with the copies of the interceptor function, the address for a specific copy to be returned in reply to the current request (A) may be computed at block 110 as follows: $A = A_0 + (i - offset) * L$; wherein A_0 is the starting address of the current memory region, L denotes the size of the interceptor function, i is the index of the current IFT element, and *offset* comprises an offset to IFT from the current region descriptor table. Then, at block 112, a next successive element of IFT may be selected as current.

The procedure may be repeated for each new function to intercept. In case the allocated memory region is full (checked at block 100), a new region may be allocated and a corresponding region descriptor updated accordingly.

The above described procedure provides an efficient mapping between a particular interceptor function copy and an intercepted function. This enables a non-intrusive interceptor function to be implemented as shown in Figure 4.

According to Figure 4, the interceptor function, once activated, may retrieve (at block 200) its current address being executed and, by searching the region descriptor table (RDT), determine the region the current address belongs to, then obtain the region's base address from a corresponding RDT entry (from a corresponding region descriptor) at block 202. A reference (i.e., an offset) to an intercepted function address table (IFT) may be fetched from the same region descriptor at block 204. An index (i) to IFT (to map the currently executed copy of the interceptor function and the original function address) may be computed at block 206 as follows:

$i = (A - A_0) / L + offset$; wherein A is the current address, A_0 is the starting address obtained from the RDT, L denotes the size of the interceptor function, and *offset* comprises an offset to the intercepted function address table.

At block 208, the arguments to pass to the original function may be prepared, and the original function may be called via a function pointer IFT[i] at block 210.

It should be noted that in some embodiments, the operations at blocks 200, 208, and 210 may be performed by calling built-in compiler functions available in most modern compilers. In order to avoid efficient address confusion when copying the compiled interceptor function code, it is recommended to call these built-in functions via function pointers.

For an exemplary, non-limiting embodiment of the present invention implemented in C language refer to Appendix A. The C code is provided for the purpose of illustration only and does not constitute a complete software binary instrumentation system. The code is to provide a pointer to an interceptor function for each function to be intercepted, to employ the same code for all interceptor functions, and, at the same time, avoid using platform-specific assembly language. One skilled in the art will recognize the option of introducing additional parameters to

the region descriptor data structure to control the filling and allocation of memory regions without departing from the scope of the present invention.

Furthermore, one skilled in the art will recognize that embodiments of the present invention may be implemented in other ways and using other programming languages.

The techniques described herein are not limited to any particular hardware or software configuration; they may find applicability in any computing or processing environment. The techniques may be implemented in logic embodied in hardware, software, or firmware components, or a combination of the above. The techniques may be implemented in programs executing on programmable machines such as mobile or stationary computers, personal digital assistants, set top boxes, cellular telephones and pagers, and other electronic devices, that each include a processor, a storage medium readable by the processor (including volatile and non-volatile memory and/or storage elements), at least one input device, and one or more output devices. Program code is applied to the data entered using the input device to perform the functions described and to generate output information. The output information may be applied to one or more output devices. One of ordinary skill in the art may appreciate that the invention can be practiced with various computer system configurations, including multiprocessor systems, minicomputers, mainframe computers, and the like. The invention can also be practiced in distributed computing environments where tasks may be performed by remote processing devices that are linked through a communications network.

Each program may be implemented in a high level procedural or object oriented programming language to communicate with a processing system. However, programs may be implemented in assembly or machine language, if desired. In any case, the language may be compiled or interpreted.

Program instructions may be used to cause a general-purpose or special-purpose processing system that is programmed with the instructions to perform the operations described herein. Alternatively, the operations may be performed by specific hardware components that contain hardwired logic for performing the operations, or by any combination of programmed computer components and custom hardware components. The methods described herein may be provided as a computer program product that may include a machine readable medium having stored thereon instructions that may be used to program a processing system or other electronic device to perform the methods. The term "machine readable medium" used herein shall include any medium that is capable of storing or encoding a sequence of instructions for execution by the machine and that cause the machine to perform any one of the methods described herein. The term "machine readable medium" shall accordingly include, but not be limited to, solid-state memories, optical and magnetic disks, and a carrier wave that encodes a

data signal. Furthermore, it is common in the art to speak of software, in one form or another (e.g., program, procedure, process, application, module, logic, and so on) as taking an action or causing a result. Such expressions are merely a shorthand way of stating the execution of the software by a processing system to cause the processor to perform an action or produce a result.

While this invention has been described with reference to illustrative embodiments, this description is not intended to be construed in a limiting sense. Various modifications of the illustrative embodiments, as well as other embodiments of the invention, which are apparent to persons skilled in the art to which the invention pertains are deemed to lie within the spirit and scope of the invention.

APPENDIX A

A C code example of instrumentation process.

The example code is to provide a pointer to an interceptor function for each function to be intercepted, to employ the same code for all interceptor functions, and, at the same time, avoid using platform-specific assembly language.

Note that in this example an additional field in the Region Descriptor (`curr_addr`) is introduced to control the filling and allocation of memory regions. Calls to the compiler's built-in functions are implemented via function pointers to avoid effective address confusion while copying the interceptor function body.

The presented code implements a statically allocated Intercepted Function address Table (IFT) and provides indices (offsets) to the IFT in Region Descriptors. One skilled in the art will recognize the option of implementing dynamic IFT allocation schemes and storing direct references (pointers) to the elements of IFT in region descriptors while staying within the scope of the present invention.

This code may be embedded in a dynamic instrumentation system and operate at runtime intercepting calls to any requested function, or constitute a part of a stand-alone binary instrumentation module operating on a given executable.

```
// Region descriptor
typedef struct
{
      void* start_addr;
      void* end_addr;
      int faddr_offset;
      void* curr_addr;
} rdesc;

// Original function address
typedef struct
{
      void* original_address;
} faddr;

// Region Descriptor Table
rdesc* rdescTab;

// Intercepted Function address Table (IFT)
faddr* faddrTab;

// Size of Interceptor Function Body (may be computed dynamically)
int fbsize;
// Size of Region (may be adjusted dynamically in case of low memory)
```

```
int rsize;

// Interceptor Function Body pointer
void* fbody = (void*)stub;

// This function receives the original address of a function to
intercept
// and returns the address of an interceptor function body
void* instrument_function(void* original_address)
{
      static int curr_offset = 0;
      static int curr_region = 0;

      if(!rdescTab[curr_region].start_addr)
      {
            rdescTab[curr_region].start_addr = malloc(rsize);
            rdescTab[curr_region].end_addr                              =
      rdescTab[curr_region].start_addr +
                                          rsize;
      rdescTab[curr_region].curr_addr                                   =
rdescTab[curr_region].start_addr;
      rdescTab[curr_region].faddr_offset = curr_offset;
      }

      faddrTab[curr_offset] = original_address;

      memcpy(rdescTab[curr_region].curr_addr, fbody, fbsize);

      void* ret = rdescTab[curr_region].curr_addr;

      rdescTab[curr_region].curr_addr += fbsize;

      curr_offset++;

      if(rdescTab[curr_region].curr_addr                                ==
rdescTab[curr_region].end_addr)
      {
            curr_region++;
      }

    return ret;
}

// Pointers to compiler's built-in functions
void* (*builtin_args)() = __builtin_apply_args;
void* (*builtin_apply)(void*, void*, int) = __builtin_apply;

void* get_current_address()
{
      return __builtin_return_address(0);
}

// Interceptor Function Body
int stub(int arg, ...)
{
      int idx;
      void* addr = get_current_address();

      for(i = 0; rdescTab[i].start_addr; i++)
```

```
        {
                if(addr >= rdescTab[i].start_addr && addr <
    rdescTab[i].end_addr)
                {
                        idx = rdescTab[i].faddr_offset +
                                (addr - rdescTab[i].start_addr) / fbsize;

                        break;
                }
        }
        void *args = builtin_args();
        void *ret = builtin_apply(faddrTab[idx], args, 104);
}
```